Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1952 – Mai 2010

On Asymmetric Progress Conditions*

Damien Imbs** , Michel Raynal*** , Gadi Taubenfeld****
damien.imbs@irisa.fr, raynal@irisa.fr, tgadi@idc.ac.il

Abstract: Wait-freedom and obstruction-freedom have received a lot of attention in the literature. These are symmetric progress conditions in the sense that they consider all processes as being "equal". Wait-freedom has allowed to rank the synchronization power of objects in presence of process failures, while obstruction-freedom (that is a much weaker progress condition) allows for simpler and more efficient object implementations.

This paper introduces the notion of asymmetric progress conditions. Given an object O in a read/write system of n processes, such a condition assumes that O can be accessed by a subset of $y \leq n$ processes only (i.e., O has y ports), and requires that O guarantees wait-freedom for x processes and obstruction-freedom for the remaining $y-x$ processes. The paper investigates the power of such a progress condition, called (y,x)-liveness ((n,n)-liveness is wait-freedom while $(n,0)$-liveness is obstruction-freedom). The main contributions of the paper are the following. (1) An impossibility result showing that $(n,1)$-liveness cannot be obtained from $(n-1,n-1)$-live objects (i.e., from any number of wait-free objects with $n-1$ ports). (2) An (n,x)-liveness hierarchy for $0 \leq x \leq n$. (3) An algorithm based on (x,x)-live objects that constructs a consensus object with an asymmetric group-based progress condition.

Key-words: Asynchronous system, Consensus number, Fault-Tolerance, Liveness, Obstruction-freedom, Process crash, Progress condition, Shared memory system, Wait-freedom.

Sur les conditions de progression asymétriques

Résumé : *Ce rapport étudie les conditions de progression asymétriques dans les systèmes à mémoire partagée.*

Mots clés : *Processus asynchrones, Simulation BG, Nombre de consensus, Calculabilité distribuée, Tolérance aux défaillances, Défaillances par crash, Algorithme de réduction, t-Résilience, Mémoire partagée, Puissance de synchronisation, Sans-attente.*

---

* This paper is an extended version of a paper to appear at PODC 2010.
** Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes
*** Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes
**** Interdisciplinary Center, Herzliya, Israel

1 Introduction

A *concurrent object* is an object that can be accessed concurrently by several processes. The implementation of such an object has first to be correct. The usual correctness condition required is *linearizability* [9] which states that the operations on an object have to appear as if they had been executed sequentially, this total order respecting their real time occurrence order. Being correct is not enough, and the implementation of a concurrent object also has to provide progress guarantees. This paper is on the definition and the study of asymmetric progress guarantees.

1.1 Progress conditions

Wait-freedom and consensus number Wait-freedom is the strongest progress condition. For a given an object, it requires that any correct process completes any operation on that object in a finite number of steps regardless of the behavior of the other processes. Wait-freedom can be viewed as starvation-freedom extended to asynchronous systems prone to process crashes. It is easy to see that a wait-free implementation cannot use locks.

A consensus object is a concurrent object that allows each process to propose a value, and guarantees that (a) every process -that proposes a value and does not crash- decides on a value (termination), (b) a decided value is a proposed value (validity), and (c) no two different processes decide distinct values (agreement). It has been shown in [7], that any concurrent object defined by a sequential specification can be wait-free implemented using wait-free *consensus objects* and atomic registers.

An important notion associated with a concurrent object is its consensus number. An object of type μ has consensus number x if x is the largest integer (or $+\infty$ if there is no such integer) such that a consensus object for x processes can be wait-free implemented from atomic registers and objects of type μ. The *wait-free hierarchy* is an infinite hierarchy of object types such that the object types at level x are exactly the object types whose consensus number is x. For example, atomic registers have consensus number 1, Test&Set objects have consensus number 2 and Compare&Swap or LL/SC objects have consensus number $+\infty$.

Obstruction-freedom Because starvation appears rarely in practice, and wait-free implementations can be complex or inefficient, a weaker progress condition called *obstruction freedom* has been proposed [8]. An obstruction-free implementation of an object guarantees that a correct process that invokes an operation returns from that invocation if it runs "long enough" in isolation (the words "long enough" are used to capture the arbitrary duration required by that process to execute the operation). While wait-freedom and obstruction-freedom are progress conditions whose definition is independent of the actual failure pattern, the second one guarantees progress only in "favorable" circumstances.

x-Obstruction-freedom is a generalization of obstruction-freedom [13, 14]. It guarantees that, for every set of processes P, $|P| \leq x$, every correct process in P returns from its operation invocation if no process outside P takes steps for "long enough". It is easy to see that x-obstruction-freedom and wait-freedom are equivalent in any n-process system where $x \geq n$. Differently, when $x < n$, x-obstruction-freedom depends on the concurrency pattern while wait-freedom does not.

1.2 Content of the paper

Both wait-freedom and obstruction-freedom are symmetric progress conditions in the sense that a given process, or a given subset of processes, is not favored with respect to the others. All processes are equal with respect to the progress condition. Their main difference lies in the fact that obstruction-freedom depends on the concurrency pattern while wait-freedom does not.

Asymmetric progress conditions This paper introduces and investigates the notion of an *asymmetric progress condition*. This investigation has two main motivations. The first is the observation that, in some applications, some processes are more important than others from the object liveness point of view. While an object can be accessed by all processes, liveness guarantees sometimes have to be stronger for a predefined set of processes. Moreover, the most important processes for a given object are not necessarily important for another object. The second motivation is theoretical. Understanding the power and the limit of asymmetric progress conditions will help us to better understand the deep nature of progress conditions, which ones are stronger than others, and which are equivalent. An ultimate goal would be to establish a "ranking" of progress conditions.

(y, x)-live objects Consider an asynchronous shared memory system of n processes. An *(y, x)-live object* is an object that (1) can be accessed by $y \leq n$ processes, and (2) satisfies wait-freedom for $x \leq y$ processes and obstruction-freedom for the remaining $y - x$ processes. The integer y defines the *size* of the object, while x defines its *liveness degree*. If $x = y = n$, the object is a classical wait-free object for the considered system.

We observe that, when we consider the spectrum from obstruction-freedom to wait-freedom, an $(n, 1)$-live consensus object is the first object stronger than an obstruction-free consensus object. More generally, from a theoretical point of view, a fundamental issue is the characterization of the power of (y, x)-live objects. To that end, the paper addresses the following questions.

- It is possible to implement a $(n, 0)$-live consensus object (i.e., an obstruction-free consensus object) for any value of $n \geq 1$ using atomic registers [8]. What happens if wait-freedom is required for one process only, and we can use wait-free consensus objects for sets of $n-1$ processes? Put another way, is it possible to implement an $(n, 1)$-live consensus object from $(n-1, n-1)$-live consensus objects and atomic registers?
- What is the consensus number of an (n, x)-live consensus object?

The first question can be reformulated as follows: is wait-free consensus for $n-1$ processes strong enough to implement a consensus object for n processes that ensures wait-freedom for only one process, and obstruction freedom for the other processes? The second question concerns the intrinsic power of (n, x)-live objects.

The paper answers the first question by proving an impossibility result, namely, it is impossible to construct a consensus object for n processes providing one process with wait-freedom and the other processes with obstruction-freedom, from wait-free consensus objects for $(n-1)$ processes and atomic registers. The paper answers the second question by showing that an (n, x)-live consensus object with $x < n$ has consensus number $x+1$, and thereby establishes a hierarchy for (n, x)-liveness.

The fault-freedom progress condition It is interesting to design algorithms that eventually achieve the goal for which they are designed at least when all processes participate and there are no failures. This is called the *fault-freedom* progress condition.

The interesting question is then the design of algorithms satisfying both obstruction-freedom and fault-freedom. The paper investigates this issue and shows that, somewhat surprisingly, it is not possible to implement a consensus algorithm for n processes that satisfies both (1) obstruction-freedom with respect to all the processes, and (2) fault-freedom with respect to even a single process, when using atomic registers and any number of $(n-1, n-1)$-live consensus objects.

Asymmetric groups of processes Let us consider a system of n processes that can access read/write registers and (x, x)-live consensus objects with $x < n$ (i.e., each consensus object is wait-free but can be accessed by a subset of x processes only). Thanks to the previous results, we know that it is not possible to design a wait-free consensus object for the whole set of n processes. It is nevertheless possible to design an algorithm that guarantees a "weak" progress condition. The algorithm is as follows. A predefined group X of x processes use an (x, x)-live consensus object to agree on a value, while the other processes wait for the value decided by the processes of X. This solution is not satisfactory for several reasons. First, it is unfair with respect to the values proposed by the $n-x$ processes that do not belong to the privileged set X (if not proposed by processes of X, their values cannot be decided). Second, and more important from a progress condition point of view, if no process of X participates, the participating processes remain blocked forever. This means that this solution offers a very weak progress property. Hence the question that comes immediately to mind: is it possible to provide the processes with a better (provable) progress property?

The paper answers this question positively by presenting an algorithm that ensures the progress condition stated just below. Observing that consensus can be solved inside any group of at most x processes, it is possible to partition the n processes into $m = \lceil \frac{n}{x} \rceil$ groups. Hence, the problem amounts to select a single value from the (at most) m values, each decided within a group. To that end, we assume that the groups are ordered from group 1 to group m, with group 1 being considered as more important than group 2, etc. It is here where asymmetry appears. The consensus algorithm that is proposed guarantees the following asymmetric progress condition. Let $y \in [1..m]$ be the first group (according to the previous total order) for which a process participates in the consensus. Then, if a *correct* process in group y participates, any correct participating process eventually decides.

The design of this algorithm is based on an original combination of consensus instances inside each group and an arbitration mechanism to select between groups. The arbitration mechanism has to be crash-tolerant. To that end, it is based on a new arbiter object for which we provide an implementation in a pure read/write crash-prone asynchronous system. Interestingly, the consensus algorithm that is obtained is also fair in the sense that, for any process, there is an asynchrony and failure pattern in which the value proposed by that process is decided.

1.3 Roadmap

The paper is made up of 7 sections. Section 2 presents the computational model and the (y, x)-liveness notion. Section 3 presents two impossibility results that define bounds on the computational power of (n, x)-live consensus objects. Section 4 presents the hierarchy associated with (n, x)-live objects, which generalizes in some sense Herlihy's wait-free hierarchy [7]. Section 5 shows that it is impossible to design a consensus algorithm that satisfies both (1) obstruction-freedom with respect to all processes and (2) fault-freedom with respect to a single process, from registers and any number of $(n-1, n-1)$-live consensus objects. Section 6 presents first the new arbiter object type, and then the consensus algorithm that guarantees the group-based asymmetric progress condition stated previously. Section 7 concludes the paper.

2 Underlying System model

Asynchronous crash-prone process model The system is made up of n asynchronous sequential processes denoted $p_1, \ldots, p_n$ (sometimes processes are also denoted p or q). A process executes a sequence of steps as defined by its algorithm. A process executes correctly its algorithm until it (possibly) crashes. After it has crashed a process executes no more steps. Given a run, a process that crashes is said to be *faulty* in that run, otherwise it is *correct*.

Communication model As indicated in the Introduction, the processes communicate via read/write registers (that every process can read or write), and (y, x)-live consensus objects.

A pair of process sets (Y, X) such that $|Y| = y$, $|X| = x$, and $X \subseteq Y$, is associated with each (y, x)-live consensus object. Only the processes of Y can access it. Such an object provides the processes of Y (only) with a single operation denoted propose(). A process can invoke it at most once; it then supplies it with the value it proposes to the consensus. Any invocation that terminates returns a value.

The properties of an (y, x)-live consensus object have already been informally stated in the Introduction. They are:

- Validity. A decided value is a proposed value.
- Agreement. No two distinct values are returned by different processes.
- Termination.
  - Wait-free termination. Any invocation issued by a correct process of X terminates.
  - Obstruction-free termination. Any invocation issued by a correct process of $Y \setminus X$ terminates if the invoking process executes alone during a long enough period of time.

It is easy to see that an (n, n)-live consensus object is a usual wait-free consensus object in a system of n processes, while an $(n, 0)$-live consensus object is an obstruction-free consensus object.

Remark Let us observe that, in the consensus problem, as soon as a value has been decided by a process, any process can decide the very same value.

Notations All shared objects are denoted with uppercase letters. Differently, local variables are denoted with lowercase letters. Sometimes the index i of process p_i is used as a subscript for its local variables.

3 Two impossibility results

3.1 Digest of the section

This section proves two impossibility results. Theorem 1 answers (negatively) the first question posed in the Introduction. Wait-freedom for only one process, and obstruction-freedom for all other processes, cannot be obtained even when, in addition to atomic registers, we are provided with the objects that are the most powerful in a $(n-1)$-process system (i.e., consensus objects that are wait-free for $(n-1)$ processes). Theorem 2 states that it is impossible to construct an $(n, x+1)$-live consensus object using any number of (n, x)-live consensus objects and atomic registers.

Theorem 1 *It is not possible to construct an $(n, 1)$-live consensus object from $(n-1, n-1)$-live consensus objects and atomic registers.*

Theorem 2 *Let x be any integer such that $1 \leq x < n-1$. It is not possible to construct an $(n, x+1)$-live consensus object using any number of (n, x)-live consensus objects and atomic registers.*

3.2 On second strongest objects

Herlihy's universality result implies that a (n, n)-live consensus is the *strongest* object in a system of n processes [7]. On another side, Gafni and Kuznetsov have shown that in a system of n processes where wait-freedom is the only progress condition, $(n-1)$-process wait-free consensus (i.e., $(n-1, n-1)$-live consensus using our notation) is the second strongest object [5] (the first one being n-process wait-free consensus). Our results show that, when we consider asymmetric progress conditions, $(n-1, n-1)$-live consensus object is *not* the second strongest object in a system of n processes. This is because an $(n, n-1)$-live consensus object is stronger than an $(n-1, n-1)$-live consensus object.

3.3 Preliminary definitions

The model of computation consists of an asynchronous collection of n processes that communicate via shared objects. An *event* corresponds to an atomic step performed by a process. For example, the events which correspond to accessing atomic read/write registers are classified into two types: read events which may not change the state of the register, and write events which update the state of a register but do not return a value. We use the notation e_p to denote an instance of an arbitrary event at a process p.

Run, implementation, prefix, extension A *run* is a pair (f, S) where f is a function which assigns initial states (values) to the objects and S is a finite or infinite sequence of events. An *implementation* of an object from a set of other objects, consists of a non-empty set C of runs, a set N of processes, and a set of shared objects O. For any event e_p at a process p in any run in C, the object accessed in e_p must be in O. Let $x = (f, S)$ and $x' = (f', S')$ be runs. Run x' is a *prefix* of x (and x is *an extension* of x'), denoted $x' \leq x$, if S' is a prefix of S and $f = f'$. When $x' \leq x$, $(x - x')$ denotes the suffix of S obtained by removing S' from S. Let $S; S'$ be the sequence obtained by concatenating the finite sequence S and the sequence S'. Then $x; S'$ is an abbreviation for $(f, S; S')$.

Enabled, indistinguishable, deterministic We say that process p is *enabled* at run x if there exists an event e_p such that $x; e_p$ is a run. For simplicity, we write xp to denote either $x; e_p$ when p is enabled in x, or x when p is not enabled in x. We say that r is a *local* register of p if only p can access r. For any sequence S, let S_p be the subsequence of S containing all events in S which involve p. Runs (f, S) and (f', S') are *indistinguishable* for process p, denoted by $(f, S)[p](f', S')$, iff $S_p = S'_p$ and $f(r) = f'(r)$ for every local register r of p. Without loss of generality, it is assumed that the processes are *deterministic*. That is, if $x; e_p$ and $x; e'_p$ are runs then $e_p = e'_p$.

The runs of an asynchronous implementation of an object (i.e., an asynchronous algorithm) must satisfy several properties. For example, if a write event which involves p is enabled at run x, then the same event is enabled at any finite run that is indistinguishable to p from x. (The proof of the theorems that follow implicitly makes use of few such straightforward properties.)

Valence, compatibility, decider The proof of the theorems considers binary consensus, i.e., if value v is proposed we have $v \in \{0, 1\}$. Let $\overline{v} = 1 - v$. It also uses the following notions. A finite run x is *v-valent* if in all extensions of x where a decision is made, the decision value is v ($v \in \{0, 1\}$). A run is *univalent* if it is either 0-valent or 1-valent, otherwise it is *bivalent*. We say that two univalent runs are *compatible* if they have the same valence, that is, either both runs are 0-valent or both are 1-valent. Finally, we say that process p is a *decider* at run x if for every extension y of x, the run yp is univalent.

3.4 Proofs of Theorem 1 and Theorem 2

Lemma 1 *In any (implementation of) obstruction-free consensus object, if two univalent runs are indistinguishable for some process p, and the state of all the objects that p can access are the same at these runs, then these runs must be compatible.*

Proof Let w and y be univalent runs such that $w[p]y$ for some process p, and the state of all the objects (local and shared) that p can access are the same at w and y. (See Figure 1(a).) Let w be v-valent. Then by the definition of obstruction-freedom, there is an extension x of w by events of p only in which p decides v. Clearly $z = y; (x - w)$ is also a run of the algorithm such that $z[p]x$. Since p decides v in z, z is v-valent. Hence, since $y \leq z$, y must also be v-valent. $\Box_{Lemma\ 1}$

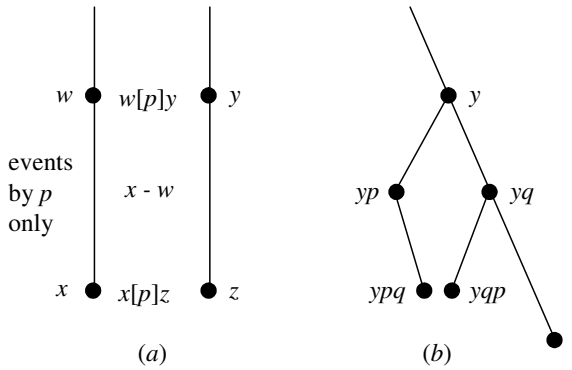

Figure 1: Runs used in proofs of Lemmas 1 and 2

Lemma 2 *Let y be a run of an algorithm implementing an obstruction-free consensus object, and p and q be two different processes such that (1) $y \neq yp$ and $y \neq yq$, (2) the runs yp and yqp are univalent and not compatible. Then, in their two next events from y, p and q are accessing the same object, and this object is not an atomic register.*

Proof Let us assume that in the last event in yp process p is accessing some object o, and in the last event in yq process q is accessing some object o'. (See Figure 1(b).)

Let us first assume that $o \neq o'$. Since the two next events from y of p and q are independent, $ypq[q]yqp$, and the values of all objects are the same in both ypq and yqp. By Lemma 1, ypq and yqp are compatible. Since ypq is an extension of the univalent run yp, it must be that yp and yqp are also compatible. A contradiction with the assumption lemma, from which follows that $o = o'$.

Let us consider that the object o is an atomic read/write register. According to the last operation issued by p in yp there are two cases.

- Case 1: In yp the last event is a *write* to o by p. Since p writes to o in its next operation from y, the value of o must be the same in yp and yqp. (Here we use the fact that the write by p overwrites the possible changes of o made by q.) Hence, $yp[p]yqp$ and the values of all the objects, which are not local to q, are the same in yp and yqp. By Lemma 1, yp and yqp are compatible. A contradiction.
- Case 2: In yp the last event is a *read* of o by p. Thus, $ypq[q]yqp$, and the values of all the objects, which are not local to p, are the same in both ypq and yqp . By Lemma 1, ypq and yqp are compatible. Since ypq is an extension of yp, it must be that yp and yqp are also compatible. A contradiction.

Thus, it must be the case that $o = o'$ and o is not an atomic read/write register. $\square_{Lemma\ 2}$

Lemma 3 *Every obstruction-free consensus object has a bivalent empty run.*

Proof By definition, the empty run with all 0 inputs must be 0-valent, and similarly the empty run with all 1 inputs must be 1-valent. Let p be an arbitrary process. Then, in any empty run with all inputs equal to v, if p executes alone it must eventually decides on v. Thus, it follows from Lemma 1 that in any empty run in which the input value of p is $v \in \{0, 1\}$, if p executes alone it must eventually decides on v. This last observation implies that every empty run in which not all inputs are 0 and not all inputs are 1 must be bivalent. $\square_{Lemma\ 3}$

Lemma 4 *For every $(n, 1)$-live consensus object there is a bivalent run x and process p such that p is a decider at x.*

Proof Let $CONS$ be an arbitrary $(n, 1)$-live consensus object which satisfies wait-freedom for process p. By Lemma 3, $CONS$ has an empty bivalent run x_0. We begin with x_0 and pursue the following *bivalence-preserving scheduling* discipline:

$x \leftarrow x_0$; $done \leftarrow false$;
repeat
  if x has a bivalent extension yp
  /* extension which involves p */
    then $x \leftarrow yp$
    /* bivalent extension of x */
    else $done \leftarrow true$ **end if**
    /* no such bivalent extension */
until $done$ **end repeat**.

Since $CONS$ satisfies wait-freedom for process p, the above procedure terminates. It follows that it terminates with some bivalent finite run x, such that for every extension y of x, the run yp is univalent, and consequently p is a decider at x. $\square_{Lemma\ 4}$

Lemma 5 *Every $(n, 1)$-live consensus object has a bivalent run y and two processes p and q such that: (1) p is a decider at y; (2) the runs yp and yqp are univalent and not compatible; and (3) in their two next events from y, p and q are accessing the same object, and this object is not an atomic register.*

Proof Let $CONS$ be an arbitrary $(n, 1)$-live consensus object, and p the only process for which it guarantees wait-freedom. By Lemma 4, there is a bivalent run x of $CONS$ such that p is a decider at x.

Let us suppose that the run xp is v-valent. Since x is bivalent, there is a (shortest) extension z of x which is $\overline{v}$-valent. (See Figure 2(a).) Let z' be the longest prefix of z such that $x[p]z'$. (See Figure 2(b).) There are two possible cases.

- Case 1: If z' is univalent we have $z' = z$. (This follows from the facts that (1) z' is the longest prefix of z such that $x[p]z'$, and (2) z is a shortest extension of x that is $\overline{v}$-valent.) Hence, z' is $\overline{v}$-valent.
- Case 2: If z' is bivalent we have $z'p = z$. (This is because $z - z'$ has a single event and this event is by p. (otherwise z' would not be the longest prefix of z such that $x[p]z'$.) It follows than $z'p$ is $\overline{v}$-valent.

Hence, in both cases, it follows from the assumption that z is $\overline{v}$-valent that $z'p$ is $\overline{v}$-valent.

Consider the extensions of x which are also prefixes of z'. (See Figure 2(c).) Since $x[p]z'$, it follows that for every y such that $x \leq y \leq z'$, $y \neq yp$. Since xp and $z'p$ are not compatible, there must exist *different* runs y and yq such that (1) $x \leq y < yq \leq z'$ and $p \neq q$; (2) yp and yqp are univalent but not compatible. It then follows from Lemma 2, that, in their two next events from y, p and q are accessing the same object, and this object is not an atomic register, which concludes the proof of the lemma. $\square_{Lemma\ 5}$

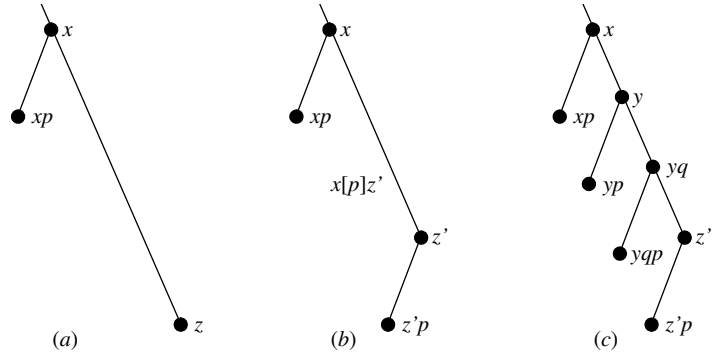

Figure 2: Illustration of runs in proof of Lemma 5

Lemma 6 *Every $(n, 1)$-live consensus object has a bivalent run y and two processes p and q such that: (1) p is a decider at y; (2) the runs yp and yqp are univalent and not compatible; and (3) in their next events from y,* all *the n processes are accessing the same object, and this object is not an atomic register.*

Proof The proof is by induction on the number of processes that access the same object. The base of the induction follows directly from Lemma 5. We assume that the theorem holds for $k < n$ processes and prove it for $k + 1$ processes.

Induction hypothesis. Every $(n, 1)$-live consensus object has a bivalent run x and two processes p and q such that: (1) p is a decider at x; (2) the runs xp and xqp are univalent and not compatible; and (3) in their next events from x, k of the n processes are accessing the same object o, and this object is not an atomic register. We denote by K the set of these k processes, and assume that p and q are in K.

Induction step. Let x be the run mentioned in the induction hypothesis, and s a process such that $s \notin K$. To prove that the claim holds for $k + 1$ processes, we show that there is an extension y of x by steps of s only such (1) p is a decider at y; (2) the runs yp and yqp are univalent and not compatible; and (3) in their next events from y, the $k + 1$ processes in $K \cup \{s\}$ are accessing the same object o and this object is not an atomic register.

Let us suppose that the run xp is v-valent and the run xqp is $\overline{v}$-valent. (See Figure 3(a).) Since x is bivalent, there is a (shortest) extension z of x by operations of s only which is univalent. We first prove that process s is accessing o in at least one of the events of the suffix $(z - x)$. Assume to the contrary that none of the events in $(z - x)$ involves s accessing o. In such a case, since in their next events from x, both p and q are accessing o, both (1) $x[p]z$ and the state of all the objects that p can access in x and z are the same (this is because the only object that p accesses in x and z is o and o is not accessed by s in z), and (2) $xq[p]zq$ and the state of all the objects that p can access in xq and zq are the same (for the same reason as before for s, and the fact that, as it is deterministic, q has accessed o with the same operation in xq and zq). As p is decider at x (and deterministic), it follows from (1) that xp and zp are compatible, and from (2) that xqp and zqp are compatible. Thus, since xp and xqp are not compatible, also zp and zqp are not compatible. But this is not possible given that zp and zqp are extensions of the univalent run z. A contradiction. Thus, at least one of the events in $(z - x)$ involves s accesses o.

Let $y \geq x$ be the longest prefix of z such none of the events in $(y - x)$ accesses o. (See Figure 3(b).) Since $ys \leq z$, y is bivalent. Furthermore, in its next events from y, process s is accessing o, and also in their next events from y, the k processes in K are accessing o. Clearly, both (1) $x[p]y$ and the state of all the objects that p can access in x and y are the same, and (2) $xq[p]yq$ and the state of all the objects that p can access in xq and yq are the same. (See Figure 3(c).) Thus, due to the same reasoning as before, xp and yp are compatible, and xqp and yqp are compatible. Thus, since xp and xqp are not compatible, it implies that also yp and yqp are not compatible. Finally, since p is a decider at x, and $x \leq y$, it follows that p is a decider at y. $\square_{Lemma\ 6}$

Proof of Theorem 1 It follows from Lemma 6 that every implementation of an $(n, 1)$-live consensus object, must use an object, say o, which all the n processes are able to access at the same run, and o is not an atomic register. Thus, it is not possible to implement an

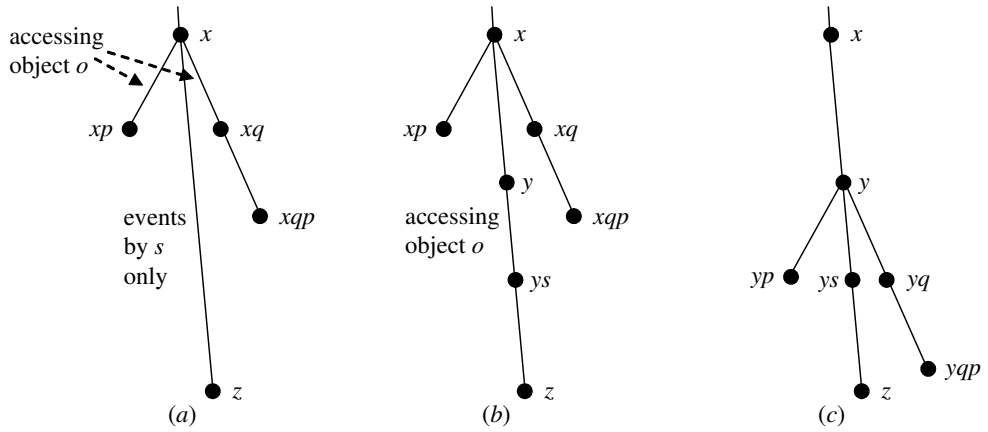

Figure 3: Runs used in proof of Lemma 6

$(n, 1)$-live consensus object using any number of $(n-1, n-1)$-live consensus objects (as each can be accessed by $n-1$ processes only) and atomic registers. $\square_{Theorem\ 1}$

Proof of Theorem 2 Assume to the contrary that there is such an implementation, say P, of an $(n, x+1)$-live consensus object from (n, x)-live consensus objects and atomic registers. As an $(n, x+1)$-live consensus object is also $(n, 1)$-live, it follows from Lemma 6 that P has a run y such that in their next events from y, all the n processes are accessing the same object, say o, and this object is not an atomic register. Thus, since o is not an atomic register, it must be the case that o is a (n, x)-live consensus object.

Let us now assume that at the end of y (just before all the processes access the consensus object o), the x wait-free processes that access object o fail, while all the other $n-x$ processes access o simultaneously. If $n-x > 1$, these processes may never run in isolation. Thus, the progress condition of o does not guarantee that any of the remaining $n-x$ processes will ever get a response from o. However, the assumption on the progress condition of the implementation P, guarantees that one of these $n-x$ processes must be wait-free. A contradiction. $\square_{Theorem\ 2}$

3.5 Objects of Common2 instead of registers

Common2 is the class of objects that have (1) consensus number 2, and (2) a wait-free implementation for any $n \geq 2$ processes using objects with consensus number 2 [2]. This class includes atomic RMW (Read/Modify/Write) registers such as Test&Set registers and Fetch&Add registers, and queues. It has recently been shown that the stack is also a member of Common2 [1].

As Common2 objects, atomic read/write registers can be accessed by any number n of processes, but differently from them their consensus number is only 1. Hence the question: is Theorem 1 still valid if we replace the atomic registers by objects in Common2 (i.e., is it or not possible to construct an $(n, 1)$-live consensus object from $(n-1, n-1)$-live consensus objects and objects in Common2)?

The impossibility still holds. This follows the simple observation that the base $(n-1, n-1)$-live consensus objects used in Theorem 1 are strictly stronger than any object in Common2. (Let $n-1 > 2$. Any wait-free consensus object for $n-1$ processes can be used to build any object in Common2 for two processes, and given 2-process Common2 objects, it is possible to extend them to obtain n-process Common2 objects [2, 6]. Differently, it is not possible to build an $(n-1)$-process wait-free object from objects with consensus number 2.)

4 The (n, x)-Liveness Hierarchy

Theorem 3 *Let x be any integer such that $1 \leq x < n-1$. The (n, x)-live consensus object type has consensus number $(x+1)$.*

Proof Let us first show that the consensus number of an $(x+1, x)$-live consensus object is at least $x+1$. Let X be the predefined set of x processes associated with the $(x+1, x)$-live consensus object, and p the process $\notin X$. As the processes of X are wait-free, it follows that there is a finite time after which no process of X is concurrent with p, which means that p can execute alone. As p is obstruction-free it also terminates. It follows that the consensus number of an $(x+1, x)$-live consensus object is at least $x+1$. Given an (n, x)-live consensus object, it is possible to restrict it to obtain an $(x+1, x)$-live consensus object. Hence, an (n, x)-live consensus object has consensus number at least $x+1$.

Let us now suppose, by way of contradiction, that an (n, x)-live consensus object has consensus number at least $x+2$. It then follows from the consensus number definition that it is possible to build a wait-free consensus object in a system of $x+2$ processes. Such an

object trivially satisfies $(x+2, x+1)$-liveness. On another side, it has been shown in Theorem 2 that an $(x+2, x+1)$-live object cannot be built using only $(x+2, x)$-live objects and atomic registers. It follows that an $(x+2, x)$-live consensus object cannot have consensus number $x+2$. Such an object has consequently consensus number exactly $x+1$. This applies also to (n, x)-live consensus objects for $n > x+2$ (by preventing the $n-(x+2)$ additional processes to participate), which concludes the proof of the theorem. $\Box_{Theorem\ 3}$

Let us recall that an $(n, 0)$-live consensus object guarantees obstruction-freedom termination. The following (n, x)-liveness hierarchy follows from Theorem 3 and the fact that both (n, n)-live consensus objects and $(n, n-1)$-live consensus objects have consensus number n. The notation $(n, x) \prec (n, y)$ means that it possible to build an (n, x)-live consensus object in a read/write shared memory asynchronous system enriched with (n, y)-live consensus objects, while the converse is not possible. The notation $(n, x) \simeq (n, y)$ means that it possible to build an (n, x)-live consensus object in a read/write shared memory asynchronous system enriched with (n, y)-live consensus objects and vice-versa.

Corollary 1 $(n, 0) \prec (n, 1) \prec \cdots \prec (n, x) \prec \cdots \prec (n, n-1) \simeq (n, n)$.

5 Impossibility of obstruction-free consensus with one fault-free process

Fault-freedom is a progress condition which guarantees that when all the processes participate and there are no failures, the algorithm (or object) eventually achieve the goal for which it is designed. Thus, a consensus algorithm which satisfies fault-freedom, must guarantee that if all the processes participate and no process fails then eventually the processes must reach an agreement. Many readers would probably agree that a correct consensus algorithm should at least satisfy fault-freedom. However, some recent important papers are not always requiring consensus to satisfy the fault-freedom progress condition.

As was shown in [8], obstruction-free consensus can be solved for any number of processes using atomic registers only. However, obstruction-freedom does *not* imply fault-freedom. Does this last possibility result justify dropping the fault-freedom progress condition? Probably not. But maybe there is a way out, maybe there is a consensus algorithm using registers that satisfies both conditions? As we prove next, there is no such algorithm.

We have already shown that it is not possible to implement a consensus object for n processes that satisfies wait-freedom for one of the processes and obstruction-freedom for all the other processes, using any number of wait-free consensus objects for $n-1$ processes (i.e., using $(n-1, n-1)$-live consensus objects) and atomic registers (Theorem 1). Is it possible to weaken the requirement that one process is wait-free and still prove a similar impossibility result? The answer is yes. Surprisingly, a similar impossibility result holds even if, instead of requiring that some process is wait-free, we only require that a *single* process is both obstruction-free and fault-free. (In the context of consensus, a process is fault-free means if all the processes participate and no process fails then eventually this process decides).

Theorem 4 *It is not possible to implement a consensus object for n processes that satisfies fault-freedom and obstruction-freedom for one of the processes and satisfies obstruction-freedom for all the other processes, using any number of $(n-1, n-1)$-live consensus objects and atomic registers.*

Proving Theorem 4 is done by modifying the proof of Theorem 1. We observe that Lemma 1, Lemma 2 and Lemma 3 are stated for obstruction-free consensus objects, and hence can be used as is. The following lemma replaces Lemma 4.

Lemma 7 *Every consensus object for n processes that satisfies fault-freedom and obstruction-freedom for one of the processes and satisfies obstruction-freedom for all the other processes, has a bivalent run x and process p such that p is a decider at x.*

Proof Let $CONS$ be an arbitrary consensus object that satisfies fault-freedom and obstruction-freedom for one of the processes and satisfies obstruction-freedom for all the other processes. By Lemma 3, $CONS$ has an empty bivalent run x_0. We begin with x_0 and pursue the following *bivalence-preserving scheduling* discipline:

$x \leftarrow x_0$; $i \leftarrow 0$; $done \leftarrow false$
repeat
  if x has a bivalent extension yp_i
  /* extension which involves p_i */
    then $x \leftarrow yp_i$; $i \leftarrow i+1 (\text{mod } n)$
    /* bivalent extension of x */
    else $done \leftarrow true$ **end if**
    /* no such bivalent extension */
until $done$ **end repeat**.

Since $CONS$ satisfies fault-freedom for some process, the above procedure terminates. It follows that it terminates with some bivalent finite run x, such that for some processes p_i, for every extension y of x, the run yp_i is univalent, and consequently p_i is a decider at x. $\square_{Lemma\ 7}$

Lemma 8 *Every consensus object for n processes that satisfies fault-freedom and obstruction-freedom for one of the processes and satisfies obstruction-freedom for all the other processes., has a bivalent run y and two processes p and q such that: (1) p is a decider at y; (2) the runs yp and yqp are univalent and not compatible; and (3) in their two next events from y, p and q are accessing the same object, and this object is not an atomic register.*

The proof of Lemma 8 is essentially the same as that of Lemma 5, replacing the reference to Lemma 4 (in the proof of Lemma 5) with a reference to Lemma 7.

Lemma 9 *Every consensus object for n processes that satisfies fault-freedom and obstruction-freedom for one of the processes and satisfies obstruction-freedom for all the other processes, has a bivalent run y and two processes p and q such that: (1) p is a decider at y; (2) the runs yp and yqp are univalent and not compatible; and (3) in their next events from y,* all *the n processes are accessing the same object, and this object is not an atomic register.*

The proof of Lemma 9 is essentially the same as that of Lemma 6, replacing the reference to Lemma 5 (in the proof of Lemma 6) with a reference to Lemma 8.

Proof of Theorem 4 It follows from Lemma 9 that every implementation of a consensus object for n processes that satisfies fault-freedom and obstruction-freedom for one of the processes and satisfies obstruction-freedom for all the other processes, must use an object, say o, which all the n processes are able to access at the same run, and o is not an atomic register. Thus, it is not possible to implement a consensus object for n processes that satisfies fault-freedom and obstruction-freedom for one of the processes and satisfies obstruction-freedom for all the other processes, using any number of wait-free consensus objects for $n-1$ processes and atomic registers. $\square_{Theorem\ 4}$

6 Consensus with group-based asymmetric progress guarantee

This section addresses the following problem. Which progress condition can be given to the processes when one wants to implement a consensus object in an n-process asynchronous read/write system enriched with (x,x)-live consensus objects (i.e., objects that wait-free solve consensus in sets of x processes).

To that end this section introduces first a new arbiter object type. Then, it states an asymmetric progress condition for the whole set of processes. This condition is based on a partitioning of the n processes into ordered groups. Finally, the corresponding n-process consensus algorithm is presented and proved correct.

6.1 The arbiter object type: definition

Definition The type arbiter provides processes with a single operation denoted arbitrate() that each process p_i can invoke at most once (on each arbiter object). Moreover, when a process p_i invokes this operation, it supplies an input parameter value $b \in \{owner, guest\}$. Let ARB be an arbiter object. If p_i invokes ARB.arbitrate($owner$), we say that it is an owner of ARB. Otherwise, it is a guest. Each invocation ARB.arbitrate() that terminates, returns a value. An object of type arbiter is defined by the following properties.

- Termination. If a correct owner invokes arbitrate(), or only guests invoke arbitrate(), or a process returns from arbitrate(), then every arbitrate() invocation by a correct process terminates.
- Agreement. No two distinct values are returned by distinct processes.
- Validity. The returned value is $owner$ or $guest$. Moreover, if no owner (resp., guest) invokes arbitrate(), the value *owner* (resp. *guest*) cannot be returned.

An implementation An implementation of an arbiter object ARB is described in Figure 4. This implementation assumes that the object has at least one and at most x owners. It uses an array $PART[owner, guest]$ (initialized to $[false, false]$), an atomic register $WINNER$ (initialized to $\perp$), and a consensus object denoted $XCONS$ that can be accessed by the x owner processes only.

When a process p_i invokes arbitrate(b), with $b \in \{owner, guest\}$, it first announces that there is at least one process of type b (owner or guest) that participates (line 01). Its behavior then depends on the fact it is an owner or a guest.

If p_i is an owner, it first agrees with the other owners on the fact that guests are or not participating (this agreement is obtained with the help of the underlying consensus object $XCONS$). If they agree on the fact that there are participating guests, p_i updates $WINNER$

```
operation arbitrate(b):
(01) PART[b] ← true;
(02) if (b = owner) then guest_win_i ← XCONS.propose(PART[guest]);
(03)                     if (guest_win_i) then WINNER ← guest else WINNER ← owner end if
(04) % b = guest % else  if (PART[owner]) then wait(WINNER ≠ ⊥) else WINNER ← guest end if
(05) end if;
(06) return(WINNER).
```

Figure 4: The arbitrate() operation of the arbiter object type (code for p_i)

to $guest$ (the guest are the winners of the arbitration). Otherwise, the owners win the arbitration and p_i updates consequently $WINNER$ to $guest$ (line 03).

If process p_i is a guest and, from its point of view, no owner participates, it considers that the guests have won the arbitration. Otherwise, p_i waits until the owners have decided which are the winners of the arbitration. (line 04). Finally, in all cases, the invoking process returns the value of $WINNER$.

Theorem 5 *Let us assume that there are at most x processes that invoke ARB.arbitrate(b) with $b = owner$. The algorithm in Figure 4 implements the* arbiter *object type.* (Proof given in Appendix A).

6.2 Consensus with group-based asymmetric progress

Let us assume that the processes are partitioned into m ordered groups. Consensus with *group-based asymmetric progress* is defined by the usual validity and agreement property, plus the following asymmetric termination property.

- Termination. If there is $y \in [1..m]$ such that (1) no process in a group $z < y$ invokes propose(), and (2) a process in group y invokes propose() and is correct, then all correct participating processes decide.

6.3 Consensus algorithm

Assuming that the underlying n-process asynchronous read/write shared memory system is enriched with (x, x)-live consensus objects, this section presents an algorithm that constructs a consensus object satisfying the group-based asymmetric progress condition stated previously. As indicated in the Introduction, the n processes are partitioned into $m = \lceil \frac{n}{x} \rceil$ groups. A process p_i determines its group by calling group(i).

Data structures The algorithm uses the following registers, arbiters and (x, x)-live consensus objects.

- $VAL[1..m]$ is an array initialized to $[\bot, \cdots, \bot]$. The aim of $VAL[g]$ is to contain the value decided inside group g.
- $GXCONS[1..m]$ is an array of (x, x)-live consensus objects (each can wait-free solve consensus in a set of x processes). The consensus object $GXCONS[g]$ is accessed by the processes of group g only. It allows them to compute the value decided inside their group (and saved in $VAL[g]$).
- $ARBITER[1..m-1]$ is an an array of arbiter objects. $ARBITER[g]$ is used by (1) the processes of the group g (which are its x owners), and (2) the processes of the groups $g+1, \ldots, m$ (which are its guests).
- $ARB\_VAL[1..m]$ is an array initialized to $[\bot, \cdots, \bot]$. $ARB\_VAL[g]$ is intended to contain the value decided by the processes in the groups $g, g+1, \ldots, m$. Hence, $ARB\_VAL[1]$ eventually contains the single value decided by all the processes.

  The value of $ARB\_VAL[g]$ is computed as follows. If the winners of $ARBITER[g]$ are its owners (i.e., the processes of the group g), then the value of $ARB\_VAL[g]$ is the value $VAL[g]$ that these processes have decided inside their group. If the winners are the guests, then the value of $ARB\_VAL[g]$ is the value already decided by these guests (i.e., the processes in the groups $g+1, \ldots, m$), that they have saved in $ARB\_VAL[g+1]$.

Process behavior The algorithm executed by a process p_i is described in Figure 5. A process participates when it invokes propose(v_i) where v_i is the value it proposes. There is no restriction on the set of values that can be proposed. It terminates when it executes the return(v) statement (where v is the value it decides). The algorithm is made up of two tasks. Task $T2$ waits for the decided value and returns it. Task $T1$ is the main task.

After having determined its group y (line 01), a process p_i invokes the consensus object $GXCONS[y]$ to learn the value decided inside its group, and it deposits it into $VAL[y]$. Then, p_i enters sequentially two competitions the aim of which is to deposit into $ARB\_VAL[1]$, a single value.

```
operation propose(v_i):
task T1:
(01) let y = group(i);
(02) VAL[y] ← GXCONS[y].propose(v_i);
(03) if (y = m) then ARB_VAL[y] ← VAL[y]
(04)              else winner ← ARBITER[y].arbitrate(owner);
(05)                   if (winner = owner)
(06)                     then ARB_VAL[y] ← VAL[y]
(07)                     else ARB_VAL[y] ← ARB_VAL[y + 1]
(08)                   end if
(09) end if;
(10) if (y > 1) then
(11)   for ℓ from (y − 1) step −1 to 1 do
(12)       winner ← ARBITER[ℓ].arbitrate(guest);
(13)       if (winner= guest)
(14)         then ARB_VAL[ℓ] ← ARB_VAL[ℓ + 1]
(15)         else ARB_VAL[ℓ] ← VAL[ℓ]
(16)       end if
(17)   end for
(18) end if.

task T2: wait(ARB_VAL[1] ≠ ⊥); return(ARB_VAL[1]).
```

Figure 5: The propose() operation (code for p_i)

- Competition #1 (lines 03-09). The aim is here to deposit a value into $ARB\_VAL[y]$. If p_i belongs to group m, there is no competition and p_i deposits $VAL[m]$ into $ARB\_VAL[m]$ (line 03). if $y < m$, p_i invokes $ARBITER[y]$ as an owner (line 04). This object arbitrates the group y on one side and the groups $y+1$ until m on the other side. If the group y wins, p_i deposits $VAL[y]$ into $ARB\_VAL[y]$ (line 06). Otherwise, p_i's group has lost this competition and consequently p_i deposits into $ARB\_VAL[y]$ the value decided by the the groups $y + 1$ to m that they have saved in $ARB\_VAL[y + 1]$ (line 07).
- Competition #2 (lines 10-18). The aim is here to deposit a value into $ARB\_VAL[1]$. To that end, once $ARB\_VAL[y]$ has been assigned a value, p_i enters a sequence of competitions first with the group $y - 1$, etc., until group 1. Let us remember that a process in group y is a guest for all arbitrations with respect to a group $\ell < y$. When competing with group ℓ to deposit a value into $ARB\_VAL[\ell]$, there are two cases. If it is a winner (which means that collectively the groups $\ell + 1$ until m have won the competition with group ℓ), p_i assigns to $ARB\_VAL[\ell]$ the value previously saved in $ARB\_VAL[\ell + 1]$ (line 14). Otherwise, the groups $\ell + 1$ until m have lost the competition with group ℓ, and p_i assigns to $ARB\_VAL[\ell]$ the value decided inside group ℓ previously saved in $VAL[\ell]$ (line 15).

It follows that $ARB\_VAL[1]$ contains either the value decided by group 1 or the value collectively decided by the groups 2 until m that they have saved in $ARB\_VAL[2]$. Similarly, $ARB\_VAL[2]$ contains either the value decided by group 2 or the value collectively decided by the groups 3 until m that they have saved in $ARB\_VAL[3]$, etc.

Remark If needed by an application, the full array $ARB\_VAL[1..m]$ could be returned as result. Let us observe that, due to asynchrony, this does not mean that two different processes p_i and p_j will receive the same array. Let $arb\_val_i[1..m]$ and $arb\_val_j[1..m]$ the array obtained by these processes. We have $arb\_val_i[1] = arb\_val_j[1] \neq \bot$, and $\forall \ell \in [2..m]$: $(arb\_val_i[\ell] \neq \bot) \wedge (arb\_val_j[\ell] \neq \bot) \Rightarrow (arb\_val_i[\ell] = arb\_val_j[\ell])$.

6.4 Proof of the algorithm

Lemma 10 *If there is $y \in [1..n]$ such that (1) no process in a group $z < y$ invokes* propose()*, and (2) a correct process in group y invokes* propose()*, then all correct participating processes decide.*

Proof Let p_i be a correct process of a group y that invokes propose() and let us assume that no process in a group $z < y$ participates. As a process decides when it executes the return() statement in task $T2$, showing the termination property amounts to show that we have eventually $ARB\_VAL[1] \neq \bot$. To that end we have to show that there a process that assigns a value to $ARB\_VAL[1]$. The proof is by contradiction. Let us assume that no process writes $ARB\_VAL[1]$.

Let p_i a correct process in group y. We first show that p_i terminates task $T1$. Due to the termination property of the consensus object $GXCONS[y]$, it follows that $GXCONS[y]$.propose() terminates (line 02). Moreover, when $y < m$, as p_i is an owner of $ARBITER[y]$ and is correct, it follows from the termination property of the arbiter type that the invocation $ARBITER[y]$.arbitrate($owner$) issued by p_i terminates (line 04). On another side, as no process in a group $z < y$ participates, it follows that only guests invoke $ARBITER[z]$.arbitrate() for $1 \leq z < y$. It consequently follows from the arbiter termination property that all these invocations issued by p_i terminate. Hence, p_i terminate task $T1$.

We now show that p_i assigns a value to $ARB\_VAL[1]$. Let us first observe that, due to the validity property of the arbiter type, p_i obtains the value $guest$ from all its invocations $ARBITER[z].\mathsf{arbitrate}()$ for $1 \leq z < y$. It follows that p_i deposits sequentially the value kept in $ARB\_VAL[y]$ into $ARB\_VAL[y-1]$, $ARB\_VAL[y-2]$, etc., until $ARB\_VAL[1]$ (lines 10-18). This means that we have to show that the value assigned to $ARB\_VAL[y]$ by p_i (at lines 06 or line 07) is a proposed value. There are two cases.

- If p_i executes line 06, it assigns to $ARB\_VAL[y]$ the value kept in $VAL[y]$ that is the value obtained from the consensus object $XCONS[y]$. As the single value decided by that object is a value proposed by a process of group y, it follows that $VAL[y]$ is updated to a proposed value.
- If p_i executes line 07, it writes the value of $ARB\_VAL[y+1]$ into $ARB\_VAL[y]$. Hence, we have to show that $ARB\_VAL[y+1]$ contains a proposed value. It follows from the validity property of the arbiter object type that, as the value returned to p_j by $ARBITER[y].\mathsf{arbitrate}()$ is $guest$, a guest process has invoked that operation. Due to the definition of guests for $ARBITER[y]$, such a guest is a process p_j belonging to one of the groups $y+1, y+2, \ldots, m$.
  - Case 1. Process p_j belongs to group $y+1$ and set $ARB\_VAL[y+1]$ to $VAL[y+1]$. In that case we trivially have $VAL[y+1] \neq \perp$.
  - Case 2. Process p_j does not belong to group $y+1$ but set $ARB\_VAL[y+1]$ to the value kept in $VAL[y+1]$. It then follows that, when it invoked $ARBITER[y+1].\mathsf{arbitrate}()$, guest p_j obtained the answer $owner$, which means that an owner process p_k (i.e., a process of group $y+1$) invoked $ARBITER[y+1].\mathsf{arbitrate}()$. It then follows from the text of the algorithm that p_k has deposited a value in $VAL[y+1]$ before invoking $ARBITER[y+1].\mathsf{arbitrate}()$. Hence, p_j writes $VAL[y+1] \neq \perp$ in $ARB\_VAL[y+1]$.
  - Case 3. Process p_j set $ARB\_VAL[y+1]$ to the value kept in $ARB\_VAL[y+2]$. Then, after having replaced y by $y+1$, let us redo the case analysis (Cases 1, 2 and 3). Let us observe that this iterative reasoning stops at the latest when we arrive at group m (because, if we arrive at $ARB\_VAL[m]$, that value has been computed by the processes of group m). Let $g \leq m$ be the group at which the iteration stops. We have then: $ARB\_VAL[y]$ has been previously set to $ARB\_VAL[y+1]$ which has been previously set to $ARB\_VAL[y+2]$, etc. until $ARB\_VAL[g-1]$ which has been previously set to $ARB\_VAL[g]$.

  It follows that eventually $ARB\_VAL[1] \neq \perp$, which contradicts the initial assumption and concludes the proof of the termination property.

$\square_{Lemma\ 11}$

Lemma 11 *No two different processes decide different values.*

Proof The proof consists in showing that at most one value can be assigned (maybe several times) to each $ARB\_VAL[y]$, $1 \leq y \leq m$.

Let us first observed that due to the agreement property of the underlying consensus objects, any $VAL[y]$ is either equal to $\perp$ (initial value) or to the single value decided from the corresponding consensus object (line 02).

The proof is by induction. For the base case, let us consider the group g such that no process in a group z, $g < z$, writes $VAL[g]$. It follows (from the agreement and validity properties of the arbiter object type) that the only value that can be assigned to $ARB\_VAL[g]$ is the value of $VAL[g]$.

Let us consider the following induction assumption: if $ARB\_VAL[y+1] \neq \perp$, that variable contains forever either the value $VAL[y+1]$ or the non-$\perp$ value kept in $ARB\_VAL[y+2]$. We show that if it is ever assigned a value, $ARB\_VAL[y]$ takes either the value $VAL[y]$ or the non-$\perp$ value kept in $ARB\_VAL[y+1]$.

Due to the agreement property of the arbiter object type, the invocations $ARBITER[y].\mathsf{arbitrate}()$ that terminate produce the same result. This result is then used to determine the value assigned to $ARB\_VAL[y]$. There are two cases.

- Case 1. The result is $winner = owner$. In that case, if an owner assigns $ARB\_VAL[y]$ it executes the lines lines 04-06 and assigns it the value $VAL[y]$. If a guest assigns $ARB\_VAL[y]$ it executes the lines 12-13 and line15, and assigns it the very same value $VAL[y]$.
- Case 2. The result is $winner = guest$. In that case, if an owner assigns a value to $ARB\_VAL[y]$ it assigns it the value in $ARB\_VAL[y+1]$ (line 07). If a guest assigns it a value, it assigns the value in $ARB\_VAL[y+1]$ (line 14). Moreover, as already shown in the proof of Lemma 10, $ARB\_VAL[y+1]$ contains then a non-$\perp$ value. On another side, it follows from the induction assumption that $ARB\_VAL[y+1]$ can take a single (non-$\perp$) value. It follows from these two observations that the property holds for $ARB\_VAL[y]$ is set to a single value.

$\square_{Lemma\ 11}$

Theorem 6 *The algorithm described in Figure 5 implements a consensus object that satisfies the group-based asymmetric progress condition.*

Proof The validity property is left to the reader. The termination and agreement properties follow from Lemmas 10 and 11, respectively.

$\square_{Theorem\ 6}$

7 Conclusion

This paper has introduced the notion of an asymmetric progress condition. An object is (y, x)-live if, while it can be accessed by only y among the n processes the system is made up of, it is wait-free for x of them and obstruction-free for the remaining $y - x$ ones. In a system of n processes, (n, n)-liveness is wait-freedom, while $(n, 0)$-liveness is obstruction-freedom.

The paper has then contributed the following results. It has first shown that it is impossible to build an $(n, 1)$-live consensus object (i.e., an n-process object that is wait-free for one process only and obstruction-free for the others) from $(n - 1, n - 1)$-live consensus objects (i.e., $(n - 1)$-process wait-free objects) and registers. This provides us with a deeper insight on the frontier separating wait-freedom and obstruction-freedom. The paper has then shown that, in a system of n processes, the consensus number of an (n, x)-live consensus object (with $x < n$) is $x + 1$. This establishes a hierarchy for (n, x)-live consensus objects.

Generalizing the first result above, we have shown that it is not possible to implement a consensus object for n processes that satisfies both fault-freedom and obstruction-freedom for one of the processes and satisfies obstruction-freedom for all the other processes, using any number of wait-free consensus objects for $n - 1$ processes and atomic registers.

Finally, after having introduced an asymmetric group-based progress condition suited to read/write systems of n processes enriched with (x, x)-live objects, the paper has presented an n-process consensus algorithm that satisfies this progress condition. This algorithm is based on a novel crash-tolerant arbiter object that is interesting by itself and could benefit other problems.

Acknowledgments

The work of D. Imbs and M. Raynal has benefited from the support of the French ANR project SHAMAN.

References

[1] Afek Y., Gafni E. and Morisson A., Common2 Extended to Stacks and Unbounded Concurrency. *Proc. 25 ACM symposium on Principles of distributed computing (PODC'06)*, ACM Press, pp. 218-227, 2006.

[2] Afek Y., Weisberger E. and Weisman H., A Completeness Theorem for a Class of Synchronization Objects. *Proc. 12 ACM symposium on Principles of distributed computing (PODC'93)*, ACM Press, pp. 159-170, 1993.

[3] Attiya H. and Welch J., Distributed Computing: Fundamentals, Simulations and Advanced Topics, (2d Edition), *Wiley-Interscience*, 414 pages, 2004.

[4] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[5] Gafni E. and Kuznetsov P., N-Consensus is the Second Strongest Object for N+1 Processes. *Proc. 11th Int'l Conference On Principle Of Distributed Systems (OPODIS 2007)*, Springer Verlag LNCS #4878, pp. 260-273, 2007.

[6] Gafni E., Raynal M. and Travers C., Test&set, Adaptive Renaming and Set Agreement: a Guided Visit to Asynchronous Computability. *26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, IEEE Computer Press, pp. 93-102, 2007.

[7] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[8] Herlihy M.P., Luchangco V. and Moir M., Obstruction-free synchonization: double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, pp. 522-529, 2003.

[9] Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[10] Lamport. L., On Interprocess Communication, Part 1: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2):77-101,1986.

[11] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

[12] Taubenfeld G., Synchronization Algorithms and Concurrent Programming. *Pearson Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.

[13] Taubenfeld G., Contention-Sensitive Data Structure and Algorithms. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer Verlag LNCS #5805, pp. 157-171, 2009.

[14] Taubenfeld G., On the Computational Power of Shared Objects. *Proc. 13th Int'l Conference On Principle Of Distributed Systems (OPODIS 2009)*, Springer Verlag LNCS #5923, pp. 270-284, 2009.

A Proof of the arbiter object type

The proof considers that there are at most x owners. Moreover, it can trivially be observed that the algorithm has no loop.

Lemma 12 *If an owner participates and is correct, then all correct participating processes terminate.*

Proof Due to the termination property of $XCONS$, no invocation $XCONS$.propose() by a correct owner can block. It follows that all correct owners terminate. For a guest, the only blocking statement of the algorithm is the wait statement at line 04 where a guest waits for a value to be assigned to $WINNER$. As (assumption) there is a correct participating owner, that owner returns from its invocation $XCONS$.propose() and assigns a value to $WINNER$ before terminating (line 03). It then follows that all correct participating guests terminate. $\square_{Lemma\ 12}$

Lemma 13 *If no owner participates, then all correct participating guests terminate.*

Proof As already indicated, a guest can block only if it executes the wait statement at line 04. A guest executes this statement only if it observes that an owner is participating (i.e., has executed $PART[owner] \leftarrow true$ at line 01). Hence, if no owner participates, we always have $PART[owner] = false$, and consequently all correct participating guests terminate. $\square_{Lemma\ 13}$

Lemma 14 *If a process terminates, then all correct participating processes terminate.*

Proof It follows from the text of the algorithm that, if a process terminates, $WINNER$ has necessarily been assigned a non-$\perp$ value. Hence, if a process terminates, all correct participating processes terminate. $\square_{Lemma\ 14}$

Lemma 15 *No two processes return different values.*

Proof The proof is a case analysis.

Case 1. If no owner participates, $PART[owner]$ remains forever equal to $false$. It follows that the guests that execute line 04 all execute $WINNER \leftarrow guest$, and consequently only the value $guest$ can be decided.

Case 2. If no guest participates, $PART[guest]$ remains forever equal to $false$, and consequently (due to its validity property), the value decided by the consensus object $XCONS$ can only be the value $false$. It follows that $WINNER$ can take the value $owner$ only.

Case 3. Both $PART[owner]$ and $PART[guest]$ are set to $true$. If $WINNER$ is never set to the value $owner$, the value $guest$ only can be decided. So, let us suppose that an owner p_i sets $WINNER$ to the value $owner$ (line 03). We show that, in that case, no process sets $WINNER$ to the value $guest$.

- Let us first consider the case of an owner that executes line 03. Due to the agreement property of the consensus object $XCONS$, a single value is decided by that object. Hence, every owner that executes line 03 sets $WINNER$ to the value $owner$.
- Let us now consider the case of any guest p_j that executes line 04. As the value decided from $XCONS$ by the owner p_i is $false$, it follows that there is an owner p_k that has proposed $false$ to $XCONS$ because it has previously read $PART[guest] = false$. This means p_k executed $PART[owner] \leftarrow true$, before reading $PART[guest] = false$. As any guest writes $PART[guest]$ before reading $PART[owner]$, it follows that any guest p_j finds $PART[owner] = true$ when it executes line 04. Hence, p_j executes the wait statement (and consequently does not update $WINNER$ to the value $guest$). It follows that $WINNER$ is never set to $guest$, which completes the proof of the agreement property. $\square_{Lemma\ 15}$

Lemma 16 *If only guests (resp., owners) participate, the value owner (resp., guest) cannot be returned.*

Proof Only an owner can assign the value $owner$ to $WINNER$ (line 03). Hence, , if no owner participates, we can never have $WINNER = owner$.

If no guest participates, $PART[guest]$ remains forever equal $false$. Hence, only the $owner$ can be proposed to the object $XCONS$. It follows that $WINNER$ can never be set to the value $guest$. $\square_{Lemma\ 16}$

Theorem 5 Let us assume that there are at most x processes that invoke ARB.arbitrate(b) with $b = owner$. The algorithm in Figure 4 implements the arbiter object type.

Proof Lemmas 12, 13, 14, 15 and 16 show that the algorithm in Figure 4 respects the properties defined in Section 6.1. $\square_{Theorem\ 5}$